When Constructors Are Called?

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass
- Since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used
- If super( ) is not used, then the default or parameterless constructor of each superclass will be executed

```
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[])
{
C c = new C();
}
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass
- The version of the method defined by the superclass will be hidden

```
class A {
int i, j;
A(int a, int b) {
```

```
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
} }
```

The output produced by this program is shown here: k: 3

- Method overriding occurs *only* when the names and the type signatures of the two methods are identical
- If they are not, then the two methods are simply overloaded

```
// Methods with differing type signatures are overloaded – not overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
```

```
// overload show()
void show(String msg) {
System.out.println(msg + k);
} }
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
} }
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

EXCEPTION HANDLING

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error
- The exception is *caught* and processed
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- Program statements that you want to monitor for exceptions are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Your code can catch this exception (using **catch**) and handle it
- System-generated exceptions are automatically thrown by the Java run-time system
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause
- Any code that absolutely must be executed before a method returns is put in a **finally** block
- This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
```

```
finally {
// block of code to be executed before try block ends
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches
- One branch is headed by **Exception**
- This class is used for exceptional conditions that user programs should catch
- There is an important subclass of **Exception**, called **RuntimeException**
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself
- Stack overflow is an example of such an error

Using try and catch

- Handling an exception manually provides two benefits
- First, it allows you to fix the error
- Second, it prevents the program from automatically terminating

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
} }
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

- A **try** and its **catch** statement form a unit
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement
- A **catch** statement cannot catch an exception thrown by another **try** statement

```
// Handle an exception and move on.
import java.util.Random;
```

```
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}}}
//Displaying a Description of an Exception
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

Output: Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
} }
```

Nested try Statements

- Each time a **try** statement is entered, the context of that exception is pushed on the stack

throw

- It is possible for your program to throw an exception explicitly, using the **throw** statement
- The general form of **throw** is shown here: **throw *ThrowableInstance***;
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**
- There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator
- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
} } }
```

Output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- A **throws** clause lists the types of exceptions that a method might throw
- Exceptions that a method can throw must be declared in the **throws** clause
- This is the general form of a method declaration that includes a **throws** clause:

type method-name(parameter-list) throws *exception-list*
{
// body of method

}

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[ ]) {
throwOne();
}
}
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
} } }
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

- **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block
- The **finally** block will execute whether or not an exception is thrown
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception
- The **finally** clause is optional
- Each **try** statement requires at least one **catch** or a **finally** clause

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
```

```
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
} }
```

Here is the output generated by the preceding program:

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```